**LABELS**

Labels are key/value pairs that are attached to objects, such as pods.

**Creating labels:**

The convention you use to develop labels is entirely up to you. The only thing to note is that the label and key you use must be unique for a given object. It is, however, good practice to use commonly used labels where people in your team can easily understand what your label means.

Commonly used key and label conventions are:

# Using environment as a key.

environment : "dev", or   
environment : "qa", or  
environment : "production"

#using release as a key.

release : "stable", or  
release : "canary"

#using app as a Key

app: "name of you app"

**Like all other Kubernetes resources you can create your label using the YAML file or using the kubectl command.**

The label specified is found in the metadata section of your YAML file

The YAML file below has a label [environment: production] and a label [app:nginx]

apiVersion: v1  
kind: Pod  
metadata:  
name: label-demo  
labels:  
environment: production  
app: nginx  
spec:  
containers:  
- name: nginx  
image: nginx:1.7.9  
ports:  
- containerPort: 80

A label can have any as many key and value specifications as you want.

metadata:  
 labels:  
 app: demo  
 tier: frontend  
 environment: prod  
 environment: test  
 version: v1.1.2  
 role: primary

**Selectors**

Using a label by itself has no effect. It is only when you use a label with a selector that you leverage the true power it has. There are two kinds of selectors. Equality-based and Set-based selectors.

Equality-based requirement

**Equality based selectors help you filter resources equal to a certain key and value. You would following operators for equality based-requirements:**

* =
* ==
* !=

Example

# this command would give us all resources with the env =prod label  
env=prodor# this command would give us all resources without the env key labelenv != prod

Let’s view a deployments labels env=dev from our previously created deployment.

kubectl get deployment -l env=dev --show-labels

As we can see, we’ve retrieved any deployment with the label[env=dev].

# Set-based requirement

Set-based label requirements allow filtering keys according to a set of values. You would the following operators for set-based requirements:

* in
* notin
* exists

**View all pods with labels env=dev and env=prod.**

kubectl get pods -l 'env in (dev,prod)' --show-labels

kubectl get pods -l 'env,env notin (dev)' --show-labels

# ReplicaController and ReplicaSet

There are different controllers in Kubernetes. The Replication Controller is used to replicate the pods. There are major advantages to this concept. Let’s look further into them.

* Let’s take an example:

You have an application up and running in Kubernetes. For some reason, the pod get crash and fails. Then you won’t be able to access the pod and you might lose data. So, to prevent this issue, developers use more than one instance running at the same time (High Availability concept). In Kubernetes, there is a concept called Replication controller to control the pods providing the high availability support.

**The Replication Controller ensures that the specific number of pods are up and running all the time.**

The Replication controller is also used for load balancing (sharing). When the number of users of the application increases, you need to add another pod to balance the load. And when it increases more till the first node’s resources are insufficient, you need to have another node with the same specifications. All these tasks are done by replicaController

**Configuring ReplicaController:**

Same as in the pod definition, the ReplicationController also have the same basic structure.

apiVersion:

kind:

metadata:

spec:

Same as in pod definition, the ReplicationController's apiVersion is also v1.

kind is ReplicationController.

Under metadata you can add the same configs we add to the pod definition. name, label etc.

The spec section is the most important part of the config file. The spec section of a definition file represents the objects created by the config file. In this case, the replicationController created pods. Hence, you need to define the pod under this section. The pod definition is written under the field named template which is a child field of spec. Under the template field in spec all you have to do is, write your pod definition excluding apiVersion and kind fields. Make sure the intends are correct.

Add the replicas field as a child field of Replication Controller’s spec and specify the number of replicas you need.

apiVersion: v1  
kind: ReplicationController  
metadata:  
name: sample-replica  
labels:  
app: myapp  
spec:  
replicas: 3  
template:  
**metadata:  
name: nginx-pod  
labels:  
app: myapp  
spec:  
containers:  
- name: nginx-container  
image: nginx**

**In this config file, you will see there are two metadata sections and two spec sections. The first metadata and spec sections are for the ReplicaController and the second metadata and spec sections are for the pod**

**Now the replication controller definition is ready to deploy. Save the file as rc-definition.yaml and execute the kubectl apply command to deploy the replication controller.**

**Once you deploy the replication controller, first the pods get deployed using the pod template.**

**To list the deployed replication controllers execute the command;**

**kubectl get replicationcontroller**

**When you delete the replica controller all the pods get deleted.**

kubectl delete replicationcontroller sample-rc